Lab 7: Macros and Subroutines

Taaruni Ananya | Session 008 | CRN: 15623

02/28/2025

Dr. Lan Gao

Introduction

Lab 7 is centered around practicing writing code with RISC-V, SNOWBALL, and VSCode as well as learning how to use macros and subroutines. Lab 7 looks into exploring how to make programs more efficient and easier to understand with text instead of registers and codes. The lab explains how to use macro, #define, and call titles. The lab also gives hands-on experience with running different scenarios and reusing code to build efficient programs.

Purpose

The purpose of this lab is to continue practicing the use of Assembly code, along with the commands and practice writing RISC-V utilizing VSCode and Terminal; this lab also works to introduce macros and subroutines, using analogies to OOP to help the student better understand the basic structure. This lab is meant to experiment with using macros and subroutines by changing existing code (such as from Lab 6) to adapt to the module.

Main Objectives

- Define macros and make programming easier for assembly language
- Define subroutines and make programming easier for assembly language
- Communicate values to macros and/or subroutines
- Perform add, subtract, NOT, NEG operations
- Learn how reference values in memory locations (such as “la a1, sum”) are difference from referencing to an immediate value (example: li)

Brief Explanation

VS Code is an integrated development environment (IDE) where Venus and RISC-V can be accessed and programmed. Macros can be executed with existing Assembly language code and made easier to interpret with #define and keywords. Subroutines are also utilized in Assembly language code to simplify and make the program easier to interpret as well. The three parts of the

lab each walk step by step through writing a regular Assembly program, adapting it to using macros, and adapting to using subroutines.

Methodology/Procedure

The procedure starts with learning how macros work. The differences and similarities are explored with code that uses macros and code that doesn't. Then, a program is written in Assembly then converted to using macros with #define. After learning how to do this, the next step is to adapt existing code from Lab 6 to using macros as well with #define and keywords. Next, subroutines are introduced and shown how they work in code, with regular Assembly code and snippets of code adapted to subroutines. Finally, code from part one is copied over and adapted to using subroutines.

Documentation

Simulating macro_example.s:

```
[tananya1@gsuad.gsu.edu@snowball ~]$ cp /home/mweeks/macro_example.s ~/Lab7_pt1.s
[tananya1@gsuad.gsu.edu@snowball ~]$ java -jar /home/mweeks/rars1_6.jar Lab7_pt1.s
Mar 07, 2025 5:45:46 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
RARS 1.6  Copyright 2003-2019 Pete Sanderson and Kenneth Vollmar

#Hello World
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
42
```

Writing a Program to Print Apple Types:

```
.macro print_string(%x)
    li     a7, 4
    la     a0, %x
    ecall
.end_macro

.macro print_char(%x)
    li    a7, 11
    la    a1, %x
    lb    a0, 0(a1)
    ecall
.end_macro
```

```
.text
main:
        li      a7, 4
        la      a0, title
        ecall

        li      a7, 4
        la      a0, apple1
        ecall

        li      a7, 4
        la      a0, apple2
        ecall

        li      a7, 4
        la      a0, apple3
        ecall

        li      a7, 93
        li      a0, 42
        ecall
```

```
.data
title:  .string "Types of apples:\n"
apple1: .string " Fuji\n"
apple2: .string " Gala\n"
apple3: .string " Granny Smith\n"
char1:  .byte 0x23
```

Apple Program Simulated:

```
[tananya1@gsuad.gsu.edu@snowball ~]$ java -jar /home/mweeks/rars1_6.jar ~/Lab7_pt1.s
RARS 1.6  Copyright 2003-2019 Pete Sanderson and Kenneth Vollmar

Types of apples:
 Fuji
 Gala
 Granny Smith
```

Lab part 2:

#define

```
#define PRINT_INT 4
#define EXIT_PROG 93
```

.macro

```
.macro print_string(%x)
    li      a7, PRINT_INT
    la      a0, %x
    ecall
.end_macro

.macro print_char(%x)
    li    a7, PRINT_CHAR
    li    a0, %x
    ecall
.end_macro
```

.data:

```
    .data
str1:   .string "Add result" # String to use
str2:   .string "Sub result" # String to use
str3:   .string "Not result" # String to use
str4:   .string "Neg result" # String to use
hexstr: .string "%x", 10 #String format to use
(hex), followed by

int1:   .word 12
int2:   .word 5
int3:   .word 1000
sum:    .word 0
```

.text:

```
    .text
    .globl  main
    .type   main, @function
```

Main:

```
main:
    li  a7, PRINT_INT
    la  a0, title
    ecall

    li  a7, PRINT_INT
    la  a0, apple1
    ecall

    li  a7, PRINT_INT
    la  a0, apple2
    ecall

    li  a7, PRINT_INT
    la  a0, apple3
    ecall

    li  a7, EXIT_PROG
    li a0, 03
    ecall
```

Lab part 3:

.data:

```
    .data
str1:   .string "Add result"  # String to use
str2:   .string "Sub result"  # String to use
str3:   .string "Not result"  # String to use
str4:   .string "Neg result"  # String to use
hexstr: .string "%x", 10      # String format to
use (hex), followed by

int1:   .word 12
int2:   .word 5
int3:   .word 1000
sum:    .word 0
```

.text:

```
    .text
    .globl  main
    .type   main, @function
```

main:

```
main:
    la x7, int1
    lw x6, 0(x7)

    la x7, int2
    lw x5, 0(x7)

    add x4, x6, x5

    la x7, sum
    sw x4, 0(x7)

    la a0, str1
    call print_string

    mv a0, x4
    call print_int

    li a0, 10
    call print_char

    li a7, 10
    ecall
```

Print_string:

```
print_string:
    li a7, 4
    ecall
    ret
```

print_int:

```
print_int:
    mv a1, a0
    li a0, 1
    ecall
    ret
```

print_char:

```
print_char:
    li a7, 11
    ecall
    ret
```

Questions:

1. **In Part 1, you may have observed that the "print_char" macro does this?**

```
li    a7, 11
la    a1, %x
lb    a0, 0(a1)
ecall
```

Why not use this instead? Explain.

```
li    a7, 11
la    a1, %x
ecall
```

The first block of code loads the character's address into a1 then proceeds to load the byte-type value into a0. The second version does not load the byte value, however this is an issue because ecall will return the address instead of the intended output.

2. **In Part 1, how many total lines are in the program? How many lines would there be if you did not use a macro? Explain how you get your answers.**

There are a total of 30 lines in Part 1, without the use of macros (including blank lines). With macros, the total would be 43 lines, with the macros taking up about 13 lines (including blank lines). I got this answer by subtracting the total number of lines with the number of lines that macros take over. Since the only change to the non-macro program is changing a few words, not lines, this would not create a change to the overall total.

3. **Why do you think the RARS and VSCode/Venus environment use different registers and values for ecall?**

   RARS and VSCode/Venus are different environments where RARS is specifically built for RISC-V while VSCode/Venus are built for general purpose, making them both use different registers and values for ecall.

4. **With a subroutine call as part of the program, how do the "step over" and "step into" buttons (for Venus) behave differently?**

   "Step over" moves through the program without moving line by line and executes the function. "Step into" on the other hand, steps into a function and goes through line by line.

5. **In Part 3, the last commands are:**

   ecall
   ret

   Why do we need "ret" when we use "ecall" to exit the program? Explain.

   "ret" is needed when we use "ecall" to exit the program because "ecall" finishes the portion's execution and "ret" returns after the function–this is needed in subroutines but not macros.

6. **Why does a subroutine need a ret instruction, but a macro does not?**

   In macro, there aren't any return addresses being stored and therefore does not need a ret instruction to jump back to the return address that is stored. In subroutines, on the other hand, the return address needs to be stored in order to jump back to that address, therefore needing a ret instruction.

7. **When you call a subroutine, how do you know if the registers will have the same values after it returns?**

   When a subroutine is called, the programmer knows if the registers will have the same values after it returns based on whether they are caller- or callee- saved (Geeks for Geeks).

8. **Suppose that it is important that your program remembers the value in register a0 after a subroutine call. What can you do outside of the subroutine to remember a0's value?**

   In order to remember a0's value outside of a subroutine, the value can be stored elsewhere before calling the subroutine by pushing it to a stack. Allocating space, saving it by pushing to that area in the stack, running the subroutine, bringing the value back, and then finally restoring the space in the stack is a method to execute this action while remembering a0's value outside of the subroutine.

9. **Suppose that you write a subroutine that other people might use. Your subroutine uses (i.e. changes) the a1 register. When someone else uses your subroutine, they may have something important in a1. What can you do inside of the subroutine so that a1's value is the same upon return as it was when the subroutine started?**

   In order to keep a1's value so that it is the same upon return as it was when the subroutine started inside of the subroutine is to save a1 inside of the subroutine. Saving space in the stack, pusheen a1 to that area, then running the function, and then bringing the value back, and finally restoring memory in the stack is a method to perform the requested action.

10. **Does using a macro make a difference for the problem of remembering register values?**

    Yes, macros are different from subroutines and each have a way of remembering register values. In macro, there aren't any return addresses being stored and therefore does not need a ret instruction to jump back to the return address that is stored. In subroutines, on the other hand, the return address needs to be stored in order to jump back to that address, therefore needing a ret instruction.

11. **The macro “print_char(char1)” works fine when you invoke it, where “char1” is defined in the data section. Suppose that you have the value in register a0 already, but you do not have it in memory, and use “print_char”. Does it work? Why or why not?**

    Yes, it works if I have the value in register a0 already, but not in memory and use “print_char”. This works because char1 is stored in memory and macro is capable of loading the memory.

12. **Suppose that you have the value in register a0 already, but you do not have it in memory. If you use a subroutine call such as “call print_int”, does it work? Why or why not?**

    No, it does not work if I have the value in register a0 already but not in memory. This does not work because macro is operating based on the parameter that it is already stored in memory. If I use a subroutine call such as “call print_int”, it does work. This is because subroutines work on the register instead of needing to store it elsewhere, call, restore, etc. as macros demand.

13. **Which approach (macro versus subroutine) is likely to generate a larger executable program, and why?**

    Macro seems to be more likely to generating a larger executable program because it is capable of storing its values and work similar to object orienting programming. It would reduce unnecessary code and make the program more efficient, both to programmers and the run time/execution of the program.

Key Code Observations

```
[tananya1@gsuad.gsu.edu@snowball ~]$ cp /home/mweeks/macro_example.s ~/Lab7_pt1.s
[tananya1@gsuad.gsu.edu@snowball ~]$ java -jar /home/mweeks/rars1_6.jar Lab7_pt1.s
Mar 07, 2025 5:45:46 AM java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
RARS 1.6  Copyright 2003-2019 Pete Sanderson and Kenneth Vollmar

#Hello World
[tananya1@gsuad.gsu.edu@snowball ~]$ echo $?
42
```

```
.macro print_string(%x)
    li      a7, 4
    la      a0, %x
    ecall
.end_macro

.macro print_char(%x)
    li    a7, 11
    la    a1, %x
    lb    a0, 0(a1)
    ecall
.end_macro
```

```
#define PRINT_INT 4
#define EXIT_PROG 93
```

```
.macro print_string(%x)
    li      a7, PRINT_INT
    la      a0, %x
    ecall
.end_macro

.macro print_char(%x)
    li    a7, PRINT_CHAR
    li    a0, %x
    ecall
.end_macro
```

d

```
print_string:
    li a7, 4
    ecall
    ret
```

```
print_int:
    mv a1, a0
    li a0, 1
    ecall
    ret
```

```
print_char:
    li a7, 11
    ecall
    ret
```